# FILCO: Flexible Composing Architecture with Real-Time Reconfigurability for DNN Acceleration


Xingzhen Chen
Brown University
xingzhen_chen@brown.edu

Jinming Zhuang
Brown University
jinming_zhuang@brown.edu

Zhuoping Yang
Brown University
zhuoping_yang@brown.edu

Shixin Ji
Brown University
shixin_ji@brown.edu

Sarah Schultz
Brown University
sarah_schultz2@brown.edu

Zheng Dong
Wayne State University
dong@wayne.edu

Weisong Shi
University of Delaware
weisong@udel.edu

Peipei Zhou
Brown University
peipei_zhou@brown.edu



## Abstract

With the development of deep neural network (DNN) enabled applications, achieving high hardware resource efficiency on diverse workloads is non-trivial in heterogeneous computing platforms. Prior works discuss dedicated architectures to achieve maximal resource efficiency. However, a mismatch between hardware and workloads always exists in various diverse workloads. Other works discuss overlay architecture that can dynamically switch dataflow for different workloads. However, these works are still limited by flexibility granularity and induce much resource inefficiency.

To solve this problem, we propose a flexible composing architecture, FILCO, that can efficiently match diverse workloads to achieve the optimal storage and computation resource efficiency. FILCO can be reconfigured in real-time and flexibly composed into a unified or multiple independent accelerators. We also propose the FILCO framework, including an analytical model with a two-stage DSE that can achieve the optimal design point. We also evaluate the FILCO framework on the 7nm AMD Versal VCK190 board. Compared with prior works, our design can achieve 1.3×~5× throughput and hardware efficiency on various diverse workloads.


## 1 Introduction

In many real-world applications, an end-to-end task must execute a variety of deep neural networks (DNNs) with fundamentally different characteristics in both computation and communication patterns. However, modern accelerators often adopt specific dataflows tailored for certain DNN types, resulting in significant performance degradation when handling workloads with diverse requirements. For example, in autonomous driving systems (ADS), there are MLPs for multilayer perceptron classification or regression [26], DeiT for image segmentation [23], MLP-Mixer for image classification [21], and PointNet for 3-D point-cloud processing [19]. Dense matrix multiply (MM) operations appear to be computation-intensive, but small matrix dimensions can shift the bottleneck to communication. Their dimensions also vary dramatically from layer to layer, resulting in large intra-model shape variance. Moreover, since different sub-tasks require varying accuracy levels, DNNs of different types and sizes are deployed, resulting in inter-model shape variance.

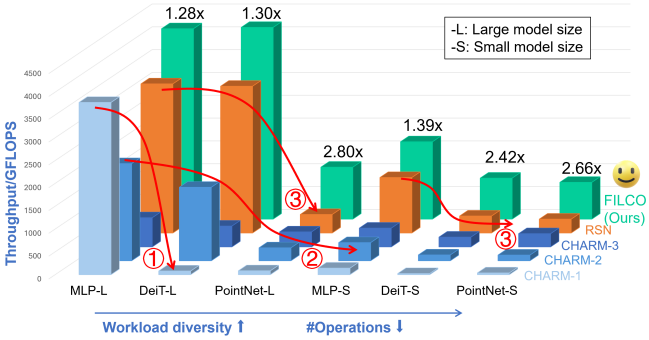



**Figure 1: Throughput comparison for different works.**

As shown in Figure 1, we profile several existing accelerators for DNN models, and we choose three models with varying degrees of diversity. In the example, we select MLP [26] to represent a DNN with low intra-model diversity, as it primarily consists of matrix multiplication layers with near-square shapes. In contrast, DeiT [23] is a Transformer-based model with a medium degree of diversity due to the different shapes of its attention and feed-forward layers. PointNet [19] exhibits the highest diversity because of its T-Net architecture. Workload diversity also exists across models of the same type but with different sizes. For example, MLP-L and MLP-S share a similar architecture yet exhibit a high degree of diversity due to inter-model variation. We also profile performance on MLP, DeiT, and PointNet models with varying sizes to showcase the capacity for inter-model diversity. CHARM-1 is one of the monolithic designs in CHARM [35], which fully utilizes the on-chip resources on the Versal platform [2]. According to profiling results, CHARM-1 can achieve high throughput for the MLP-L model due to efficient resource utilization for large and non-diverse MM. However, when the workload switches to the model with a higher degree of diversity (DeiT-L) or with a smaller size (MLP-S), the throughput degrades rapidly (①). Although CHARM-1 fully utilizes resources, it has to pad operand matrices to the fixed on-chip buffer size when executing smaller and diverse workloads, inducing much communication and computation overhead. Designing two diverse accelerators to suit diverse operand shapes can be a solution, as shown in CHARM-2 and CHARM-3. From the profiling results, such a design methodology can achieve steady performance degradation for smaller model sizes and higher degrees of diversity (②). However, since they reallocate a small portion of resources for a small accelerator, CHARM-2 and CHARM-3 cannot achieve



*DAC '26, Long Beach, CA, USA*



ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804189

as high throughput as CHARM-1 can in the large and non-diverse models (MLP-L). Since the resource partition must be determined before runtime and lacks reconfigurability, the trade-off always happens in a series of static accelerator designs.

Existing work [16] attempts to address this issue by instantiating multiple sub-accelerators for dedicated workloads, but they only support limited dataflow and lack customization for different workloads. Other existing works have tried to solve it in an overlay fashion, e.g., RSN [24]. Since RSN can flexibly map operand matrices to on-chip buffers and concatenate computation tiles for diverse workloads, it can alleviate this problem to some extent. However, RSN is limited by its static on-chip matrix shape and fixed computation tile size across computation cores. As a result, it can sustain high efficiency only when the model size is large and the degree of diversity remains relatively low. As shown in Figure 1 ③, RSN can sustain better throughput from MLP-L to DeiT-L than CHARM, but when the model size becomes smaller and the workload diversity increases, RSN suffers from a sharp drop in performance.

In order to unlock the full flexibility for hugely diverse workloads, it is essential and non-trivial to explore the flexibility and reconfigurability on both the computation side and the communication side. On the computation side, this requires designing computation logic with runtime-flexible computing tile sizes to eliminate additional overhead when launching each computation core in diverse workloads. On the communication side, in order to avoid unnecessary off-chip access, it requires designing a flexible on-chip memory that can be configured to store diverse operand shapes without extra padding overhead. In addition, flexible mapping between operand matrices and on-chip memory is essential to improve the alignment between workloads and the underlying hardware. To address these challenges, we propose **FILCO**, a flexible composing architecture with real-time reconfigurability for DNN acceleration. We summarize our contributions as follows:

- On the hardware side, we propose an AIE programming method that supports fine-grained, runtime-flexible parallelism patterns and an on-chip memory management method to avoid communication overhead.
- On the algorithm side, we formulate DSE into a two-stage optimization flow and apply MILP to search for the optimal design point, as well as a GA heuristic to reduce DSE search time.
- We propose the FILCO framework that can take DNN models, platform information, and DDR profiling results as input, and generate the ready-to-run binary files as output. We deploy our framework and conduct experiments on diverse workloads, which can achieve 1.3×~5× gains compared with existing works.

## 2 FILCO Architecture

In this section, we first introduce the overview of FILCO architecture in Section 2.1. Then, we discuss the three proposed hardware methodologies in Sections 2.2, 2.3, and 2.4. In Section 2.5, we explain the control flow and instruction set in FILCO.

### 2.1 Architecture Overview

The FILCO hardware architecture overview is shown in Figure 2. It is mainly composed of a data plane, responsible for computation and data movement, and a control plane, which manages instruction generation and execution scheduling. In the data plane, we partition

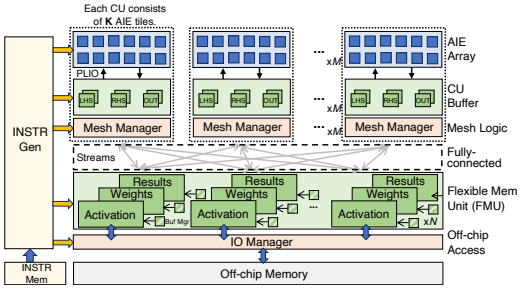



**Figure 2: FILCO hardware architecture.**

the on-chip resources into Compute Units (CU), Flexible Memory Units (FMU), and IO Manager (IOM). Each Compute Unit is featured with an AI Engine (AIE) array, a CU Buffer, and a Mesh Manager, and is responsible for handling the compute-intensive workloads. The Flexible Memory Units explores the data reuse by allocating on-chip buffers on the Programmable Logic (PL). Additionally, the IO Manager is to handle the data communication between FMUs and off-chip memory. The execution of each unit in the data plane is controlled by the instruction sets proposed by FILCO. In the control plane, the Instruction Generator loads instructions from off-chip Instruction Memory and dispatches them to each function unit according to the instruction header. Each function unit first receives and decodes the instruction, then executes it according to the control signals in the instructions.

In our proposed FILCO architecture, there are $N$ FMUs and $M$ CUs. Each CU consists of $K$ AIEs. The function units are connected through multi-level streaming hierarchies. For the off-chip communication, we design IO Managers that enable different FMUs to access a unified memory space. For the on-chip communication, we apply a fully-connected stream topology between the FMUs and CUs to achieve the maximum flexibility that can adapt to diverse workloads. Within each CU, there is a Mesh Manager to handle the mesh-in and mesh-out logic control.

To sustain both off-chip and on-chip bandwidth, proper buffer partitioning is required to avoid bank conflicts. We adopt wide port widths for off-chip access with cyclic partitioning, while using block partitioning to match the tiled MM execution of AIE and prevent on-chip buffer conflicts. To decouple partitioning conflicts between the AIE and IO Manager, we introduce a hierarchical on-chip memory, including CU Buffer and FMU. In FILCO, CU Buffers are sized to match the maximum AIE tile and use block partitioning, while FMU size depends on available on-chip buffers and adopts cyclic partitioning for wide ports. Next, we detail the hardware design methodologies in the following sections.

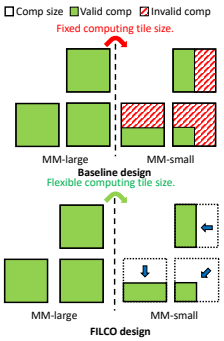
### 2.2 Flexible Computation Parallelism

In FILCO design, we assign the computation-intensive workloads to AIE. In order to achieve high efficiency on diverse workloads without reloading bitstreams at runtime, we propose a flexible AIE programming method that can switch between different execution modes at runtime to achieve flexible parallelism patterns.

Existing works either use static AIE instructions for peak efficiency or finite instruction blocks for mode switching. However, static instructions lead to performance degradation on diverse workloads. As shown in Figure 3(b), the compute tile size remains fixed across iterations. While large workloads fully utilize parallelism

```
1 void singleAIE_kernel(input_buffer&
      in0, input_buffer& in1,
      output_buffer& out0) {
2   // Load instr vec from local buf
3   vector<3> Instr = load_v<3>(in0);
4   // Assign flexible loop boundary
5   int bound_i = Instr.get(0);
6   int bound_k = Instr.get(1);
7   int bound_j = Instr.get(2);
8   // Main loop
9   for(int i=0; i < bound_i; i++){
10    for(int j=0; j < bound_j; j++){
11      for(int k=0; k < bound_k; k++){
12        //Atomic operation for 2x8x8
13        ...}}}}
```

(a)AIE kernel pseudo-code

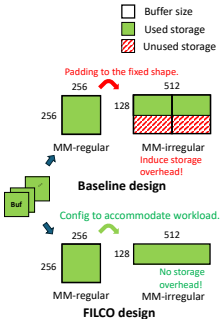



(b)Flexible computing tile sizes

**Figure 3: Flexible parallelism and single AIE programming.**

```
1 void FMU_kernel(stream &Instr,
2 stream &IOM2FMU,stream &FMU2CU0,..)
3 { //Instantiate 1-D double buffer
4   buf0[buf_size]; buf1[buf_size];
5   //Cyclic partition
6   #pragma var=buf0 buf1 type=cyclic
7   while(1){
8     //Load instr in control plane
9     instr = Instr.read();
10    if(instr.ping){
11    //Diff buf view based on instr
12    buf0_recFromIOM(instr.count,..);
13    buf1_sendToCU(instr.startRow,instr.
            endRow,instr.startCol,instr.
            endCol,..);
14    }else{.../*pong operations*/}}}
```

(a)FMU pseudo-code




(b)Flexible memory views

**Figure 4: Flexible on-chip memory views.**

(green blocks), smaller workloads require padding, resulting in significant invalid computation (red blocks). Designing finite instruction blocks helps to mitigate the invalid computation, but it has significant limitations in practice. There are only 16KB of instruction memory in each AIE, and the instruction size for computing MM with a tile size of 32x32x32 is more than 4KB.

In FILCO, to improve flexibility and efficiency, we pack each 2×8×8 tiled matrix multiplication into an atomic operation to maintain high VLIW efficiency (Line 13 in Figure 3), and define nested `for` loops with dynamic boundaries to enable flexible tile sizes (Lines 10~12). The loop boundaries are provided through input ports (Lines 3~7). At runtime, tile sizes are configured by issuing different instructions, as shown in Figure 3(b). For large workloads, the outer loop boundaries are adjusted to fully utilize compute resources. For small workloads, reconfiguration reduces tile sizes to match workloads, significantly minimizing unnecessary computation (green blocks in FILCO).

## 2.3 Flexible On-chip Memory View

As shown in Figure 4(b), existing accelerators typically use an N-dimensional on-chip buffer that matches operand dimensions. However, such static buffer views lead to poor storage utilization for diverse workloads in practice. For example, as presented in the baseline design in Figure 4(b), assume that we have a 2-D matrix with size 256x256 to be stored on-chip. In existing static accelerator designs, they instantiate a static buffer shape of 256x256 to perfectly match the workload shape, which can achieve a high efficiency in certain static workloads (green block). However, when they are

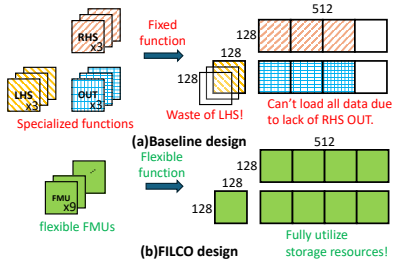



**Figure 5: Flexible on-chip memory functionality.**

required to handle diverse workloads, e.g., 128x512 matrix shapes, such a static design method induces much storage overhead, and only achieves 50% efficiency due to unnecessary padding (red block). Actually, the two diverse matrices have the same data size, which can definitely be stored in one buffer.

Therefore, proposing a flexible on-chip memory that is able to switch between different buffer views can improve efficiency. As shown in Figure4(a), FILCO instantiates multiple parallel FMUs containing a 1-D addressing double-buffer and an instruction decoder (Line 5 in Figure 4). When the system is launched, each FMU instance starts to execute simultaneously and decode instructions from the control plane (Lines 9-11). In the receive stage, FMU only receives a given number of elements that is determined from instructions (Line 14). In the send stage, FMU decodes the tile information to address it from 1-D indexing and sends it to the correct downstream function units through pre-routed streams (Line 15). After enabling such flexible on-chip memory views, FMU can be viewed as diverse shapes to perfectly match workloads, which can reduce much communication overhead and improve storage efficiency.

## 2.4 Flexible On-chip Memory Functionality

To maximize on-chip memory reuse, FILCO introduces FMUs with flexible memory allocation. Unlike prior designs that statically allocate weight and activation buffers at compile time, FMUs can be dynamically configured at runtime to support diverse workloads. In practice, workloads may have one dimension much larger than others in MM. Static buffer allocation cannot handle this efficiently, as larger dimensions require more storage for certain operands, exceeding buffer limits and preventing full data loading (Figure 5 (a)). In FILCO, to avoid the limitation of specifying buffers for a certain operand, our proposed FMU is able to be configured with different functionalities for operands or results based on control instructions. As shown in Figure 2, we connect each FMU instance with the IO Manager and CUs using pre-routed streams to construct the flexible dataflow plane. Each FMU has an independent instruction decoder to configure according to the instructions. For diverse workloads, FILCO can maximize data reuse as long as the total data size of operands and results matrices does not exceed resource constraints in Figure 5 (b).

## 2.5 Instruction Set

In this section, we elaborate on FILCO control flow and instruction set design. FILCO distinguishes between static and runtime parameters. Static parameters are fixed before compilation, such as the number and capacity of FMUs/CUs and AIE connections within a CU. Runtime parameters can be configured at execution time through instruction decoding. For example, FMUs can adapt

**Table 1: Instruction sets for function units.**

| Function Unit | Instruction Items |
|---|---|
| Instr Generator | is_last, des_unit, valid_length |
| IOM Loader | is_last, ddr_addr, des_fmu, M, N, start_row, end_row, start_col, end_col |
| IOM Storer | is_last, ddr_addr, src_fmu, M, N, start_row, end_row, start_col, end_col |
| FMU | is_last, ping_op, pong_op, src_cu, des_cu, count, start_row, end_row, start_col, end_col |
| CU | is_last, ping_op, pong_op, src_fmu, des_fmu, count |

to different data sizes, with patterns switched by decoding a few bytes of instructions. Therefore, an instruction set enabling runtime configurability is essential for supporting diverse workloads.

Table 1 lists the instruction sets for different function units. In general, Instruction Generator loads the instruction header from off-chip memory, which contains the valid instruction length in the instruction sequence and the destination units. Based on the DDR address from the instructions, the IO Managers achieve high DDR bandwidth by issuing AXI transactions with large burst length. The FMUs use the correct src/des units and 1-D addressing control information provided by the instructions to gather and scatter data, thereby maximizing the on-chip data reuse. CUs perform computation operations and are responsible for loading operands from correct FMUs and storing results to correct FMUs, as well as achieving high computation resource efficiency.

## 3 Analytical Model

In this section, we first introduce the FILCO framework overview and the proposed two-stage design space exploration (DSE) in Section 3.1. We formulate the DSE problem into an MILP formulation in Section 3.2 to guarantee the optimality. To reduce search time, we propose a Genetic Algorithm–based heuristic in Section 3.3.

### 3.1 Two-stage Design Space Exploration

Figure 6 shows an overview of the FILCO framework. FILCO takes DNN models, platform information, and DDR profiling results as input. It then performs the automated optimization flow and code generation. Finally, it launches the backend compilers to generate the ready-to-run binary files. In the first stage, Runtime Parameter Optimizer performs a brute-force search on every layer to find the optimal runtime dataflow, as well as a table with the optimal latency under the constraints of FMU and CU. In the second stage, the Schedule Optimizer searches for the optimal schedule timeline, based on the recorded table in the first stage, while guaranteeing that the resource requirements are always under hardware resource constraints. After two-stage DSE optimization, FILCO generates a valid schedule as well as customized dataflow for each DNN layer. Then the Code Generator and Instruction Generator emit the HLS codes and instruction sequences for backend compilers based on the scheduling timeline and the recorded runtime dataflow information.

### 3.2 MILP Formulation

FILCO proposes a two-stage optimization algorithm to solve the coupled mapping and scheduling problem. It ensures optimality by exhaustively enumerating possible layer-to-accelerator candidates, and then formulating an MILP to solve the scheduling stage. After

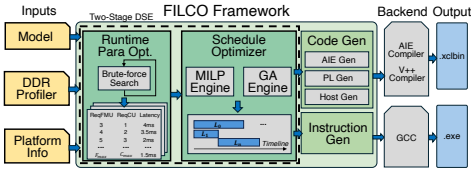



**Figure 6: FILCO framework overview.**

enumerating the possible mapping candiates in Stage 1, the scheduling optimization flow in FILCO can be formulated as the following problem [5, 34]: Given a Directed Acyclic Graph (DAG) for a workload, each node stands for one layer ($L_i$), and each edge stands for the dependency between two layers ($P_{i,j} = 1$, if $L_j$ depends on $L_i$). For i-th layer, there are multiple candidate execution modes, and for k-th mode, Runtime Parameter Optimizer records the required number of FMU ($f_{i,k}$), the required number of CU ($c_{i,k}$), the optimal runtime parameters and the optimal latency ($e_{i,k}$). For platform resource constraints, there are a total of $F_{max}$ FMUs and $C_{max}$ CUs. Our MILP formulation aims to minimize the overall execution time while preserving application dependencies and ensuring that the hardware utilization remains within the resource constraints.

We define the decision variables as follows: $A_{i,m}$ and $B_{i,m}$ are binary variables, which are equal to 1 if $L_i$ uses the m-th FMU or CU, respectively; $M_{i,k}$ is also binary variable and is equal to 1 if $L_i$ executes in k-th mode; $S_i$ and $E_i$ represent the start time and end time for layer $L_i$, respectively. After defining decision variables, we derive the objective function and constraints.

In a feasible schedule solution, each layer $L_i$ only executes in one certain mode, thus, we have

$$\forall i : \sum_k M_{i,k} = 1 \tag{1}$$

For the two consecutive layers with $P_{i,j} = 1$, the schedule timeline should meet data dependency constraints, i.e., the start time of $L_j$ should be later than the end time of $L_i$, and the end time for $L_i$ is the start time plus the latency of the selected execution mode:

$$\begin{gathered} \forall (i,j) \in DAG, P_{i,j} = 1 : S_j \geq E_i \\ E_i = \sum_k M_{i,k} \times e_{i,k} \end{gathered} \tag{2}$$

For the two non-consecutive layers with $P_{i,j} = 0$, since each FMU or CU can only execute one layer at one time, if two layers are allocated on the same FMU or CU, the execution in timeline should not overlap. To represent the overlap condition, we define a binary variable $O_{i,j}$ where

$$O_{i,j} = \begin{cases} 1, & \text{if } S_i - E_j < 0, \\ 0, & \text{if } S_i - E_j \geq 0. \end{cases}$$

To linearize the conditional constraints for the MILP formulation, we introduce a large enough integer $\phi$ and rewrite the constraints in linear form.

$$S_i - E_j < \phi \times (1 - O_{i,j}) \qquad S_i - E_j \geq -\phi \times O_{i,j} \tag{3}$$

According to the definition of $O_{i,j}$, if schedules of $L_i$ and $L_j$ are overlap in timeline, meaning that $S_i < E_j$ and $S_j < E_i$, then $O_{i,j} = O_{j,i} = 1$. Therefore, for any pair of layers with $P_{i,j} = 0$ that occupies the same FMU or CU, we have

$$\begin{gathered} \forall (i,j) \in DAG, \forall m \in FMU, P_{i,j} = 0 : A_{i,m} + A_{j,m} + O_{i,j} + O_{j,i} \leq 3 \\ \forall (i,j) \in DAG, \forall m \in CU, P_{i,j} = 0 : B_{i,m} + B_{j,m} + O_{i,j} + O_{j,i} \leq 3 \end{gathered} \tag{4}$$

If $A_{i,m} + A_{j,m} + O_{i,j} + O_{j,i} > 3$, it implies $A_{i,m} = 1, A_{j,m} = 1, O_{i,j} = 1, O_{j,i} = 1$. According to the definition of $A_{i,m}$ and $O_{i,j}$, for the pair

of $L_i$ and $L_j$, both of them are allocated to m-th FMU, and their schedules overlap, which conflicts with resource constraints.

Besides, in order to meet the resource requirements of each candidate mode, we should guarantee that the sum of utilized FMU or CU is equal to the allocated FMU or CU:

$$\forall i: \sum_m A_{i,m} = \sum_k M_{i,k} f_{i,k} \qquad \forall i: \sum_m B_{i,m} = \sum_k M_{i,k} c_{i,k} \tag{5}$$

The object is to find the feasible schedule with optimal latency:

$$\min T \quad \forall i: T \geq E_i \tag{6}$$

Combining Equations 1–6, we can formulate the scheduling optimization flow in FILCO DSE into an MILP problem.

## 3.3 Genetic Algorithm

The main challenge in the two-stage optimization flow is the extremely large design space for scheduling. For example, the complexity for enumerating only the candidate mode selection is $O(m^n)$, where $m$ is the number of candidate modes and $n$ is the number of layers in the workload DAG. To solve this problem, we propose a Genetic Algorithm-based heuristic search for Scheduling Optimizer in FILCO DSE. In our Genetic Algorithm, we first encode our decision variables into one chromosome and initialize the population with encoded chromosomes. Then we set a maximal iteration time, and within each iteration, every chromosome in the population applies crossover and mutation to generate a new generation. It then evaluates the fitness score of the child's chromosome, saves the one with the best fitness value, and iterates for the next generation.

In the FILCO framework, each design point is defined as a chromosome with $2N$ decision variables, where $N$ is equal to the number of layers. The first $N$ variables are defined as Encode[N], which are real numbers between 0 and 1; The second $N$ variables are defined as Candidate[N], which are integers between 0 and (#Can − 1), where #Can is the number of possible execution candidates. In the rossover and mutation stages, we apply a random selection strategy. In the decode and evaluation stage, we apply a dependency-aware decoding method to make sure that the dependency is resolved.

Assume there is a child chromosome encoded as Figure 7(a) and the workload DAG as Figure 7(b). According to the data dependency, we can append $L_0$ and $L_1$ to the Resolved List in Figure 7(c), then we search for the layer with a smaller chromosome Encode[$i$]. In this case, Encode[1] is smaller than Encode[0], thus, we append $L_1$ in the Schedule Order List. Then we iteratively check the dependency resolution and append the layer with the smaller Encode[$i$] to generate a complete Schedule Order List. Then, we start to schedule layers on the timeline following the order shown in Figure 7(d), to explore the parallel execution under resource constraints. After generating the schedule timeline, we can get the makespan for the chromosome and evaluate its fitness, keep the chromosomes with the best fitness, and iterate for the next generation.

# 4 Experiment Results

In this section, we first illustrate the single AIE efficiency in Section 4.1, and design a series of MM workloads to show FILCO performance robustness on diverse workloads in Section 4.2. We also conduct ablation experiments on the end-to-end throughput from BERT-32 to BERT-512 in Section 4.3. We evaluate our DSE algorithms in Section 4.4. For baseline works, we apply the CHARM framework to obtain experiment results, and we build an in-house

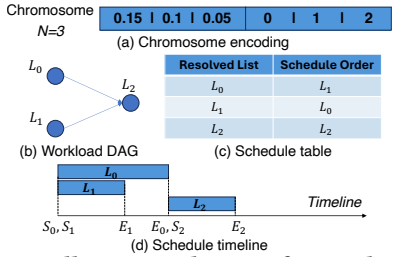



**Figure 7: Illustration diagram for GA decoder.**

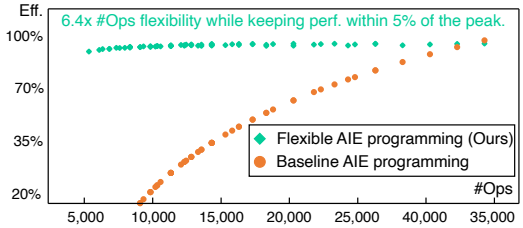



**Figure 8: Single AIE efficiency under #operations variation.**

RSN analytical model for experiments, since RSN does not provide an analytical model. All experiments are conducted on VCK190 [2] with 150MHz on PL and 1GHz on AIE. AMD/Xilinx Vitis version 2023.1 is used as the compilation backend tool. For DSE, we leverage CPLEX and Pymoo to solve MILP and GA problems, respectively.

## 4.1 Single AIE Kernel Efficiency Comparison

In this section, we illustrate the single-AIE computational efficiency gains of our proposed Flexible AIE programming using the FP32 MM benchmark of varying sizes. We apply the AIE intrinsics [3] to program the single AIE kernel and measure the execution cycles on the Versal ACAP AI Engine System C simulator [4]. As shown in Figure 8, we evaluate the MM size across from 8x24x16 to 32x32x32 with the granularity of an atomic operation, i.e., 2x8x8. The results show that our design can sustain diverse MM sizes ranging from 14x24x16 to 32x32x32, achieving over a 6× variation in operation counts with only a 5% efficiency loss. In contrast, static AIE programming induces much more overhead due to data padding, causing a huge performance drop when the MM size is small.

## 4.2 Performance Comparison on Diverse MM

In this section, we design a series of Transformer-based workloads with varying *sequence length*, *number of heads*, *head dimension*, and *MLP ratio*. Then, we categorize them according to the number of operations and inter-layer diversity, as shown in Figure 9. When the operation count is large and the diversity degree is small, both CHARM and RSN can sustain a relatively high performance. As the diversity degree increases, due to the larger MM shape variance, the performance drops sharply in CHARM, since it has to pad operand matrices to fit on-chip buffer shape, which incurs much more off-chip communication overhead. RSN can support efficient resource utilization but only when the operation counts are large. This is because RSN can flexibly map operand matrices into different on-chip memory units only with fixed matrix shapes. As the MM size decreases, each memory unit incurs extra data padding, resulting in under-utilized computation resources and communication overheads for padded operands. FILCO can sustain a large range of workload diversity since it can flexibly adjust the computation tile size and on-chip memory views to perfectly match operands and

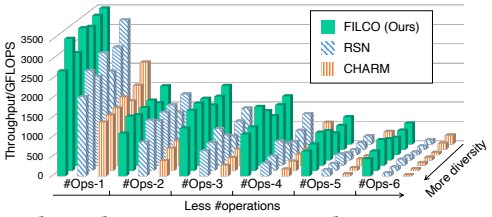



**Figure 9: Throughput comparisons on diverse MM workloads.**

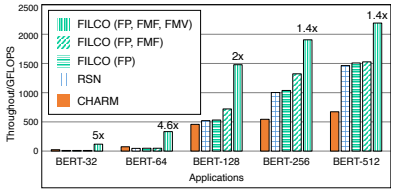



**Figure 10: End-to-end performance of realistic BERT models. FP: flexible parallelism; FMF: flexible memory functionality; FMV: flexible on-chip memory views.**

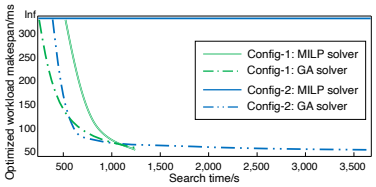



**Figure 11: Comparison of search time for MILP and GA solver.**

avoid resource under-utilization. In large MM workloads with a low degree of diversity, FILCO can achieve 1.3x throughput gains, and in small MM workloads with higher diversity, FILCO can achieve more than 5x throughput gains compared with RSN and CHARM.

## 4.3 Performance for Realistic Workloads

To evaluate the FILCO adaptability under diverse workloads and showcase the effectiveness of design methodology, we apply the FILCO framework to a series of BERT models: BERT-32, BERT-64, BERT-128, BERT-256, and BERT-512. As shown in Figure 10, we evaluate three FILCO designs with different enabled features. For the small BERT applications, limited by a low CTC ratio, the communication time dominates the end-to-end throughput. Since RSN and CHARM are not able to adjust the tile size in one memory unit, it induces much bandwidth waste due to data padding. FILCO (FP, FMF) and FILCO (FP) are also inefficient, as they load many padded operand matrices, incurring significant communication overhead. While FILCO (FP, FMF, FMV) can efficiently handle it since FMUs can flexibly adjust the buffer view to perfectly match the small matrices without data padding. In the large BERT applications, CHARM and RSN can achieve a relatively high efficiency. However, since RSN does not explore the design space for diverse parallelism patterns, and the functionalities of on-chip memory are fixed before compilation time, this leads to a sub-optimal design point. After enabling these features, FILCO can achieve better design points compared with RSN and CHARM.

## 4.4 Evaluation for DSE Search Efficiency

We evaluate the search time to demonstrate the efficiency and scalability of our proposed MILP and GA-based DSE methods. As shown in Figure 11, we compare the MILP and GA search time under two task sets. *Config-1* represents a workload with 50 layers, each with 50 candidates, whereas *Config-2* is a workload with 50 layers, each with 5000 candidates. In the small task set, the GA algorithm achieves a near-optimal solution with a faster convergence rate than MILP, with only about a 3% optimality gap. In the large task set, GA is able to produce a good design point within 10 minutes, whereas MILP fails to obtain a valid solution even after one hour. Therefore, for small workloads, FILCO can obtain optimal schedules using the proposed MILP formulation. For larger workloads, FILCO GA delivers near-optimal solutions with significantly shorter search time, showing strong efficiency and scalability.

# 5 Related Work

Existing works have two trends to explore hardware efficiency for diverse workloads: fixed-dataflow accelerators [6, 8, 10, 11, 14, 15, 17, 18, 30, 31, 33, 35, 37] and overlay-based accelerators [1, 12, 22, 24, 28, 32]. Fixed-dataflow accelerators are efficient in static and lower-diversity scenarios, since they must explicitly consider the datapath for every possible execution pattern, inducing much overhead and resource under-utilization. CHARM [35] designs a two-diverse accelerator, but they still require extra data padding when the hardware does not match workloads. DNNExplorer [33] designs a pipeline-generic hybrid architecture, which also fixes the datapath and does not alleviate the problems. SSR [37] and EQ-ViT [10] cannot handle applications with large model sizes, limited by their execution patterns. FlightVGM and FlightLLM [18, 31] suffer from performance degradation on applications except for LLM. Herald [17] provides a coarse-grained flexibility, and spatially implementing accelerators induces performance degradation when batch numbers are small.

Overlay-based accelerators design a flexible datapath and apply token-based control to enable different datapaths for diverse workloads. InTAR [12] designs a runtime fixed datapath, and the functionalities of each PE cannot be reconfigured. FEATHER [22] mainly focuses on data layout reordering. NSFlow [28] is limited by on-chip memory management, as the dataflow between PEs and on-chip memory is static at runtime. RSN [24] alleviates the problem to some extent, but their computation parallelism is static, and on-chip memory management is not flexible enough, incurring much performance degradation due to computation under-utilization and communication overhead.

Memory management is key to improving hardware utilization. GraDMM [25] proposes a dynamic memory management library for HLS-based FPGAs. Beyond on-chip techniques, several works [7, 20, 27, 29] extend accelerator memory hierarchies to incorporate off-chip DRAM and storage systems to address the rapid growth of model sizes. Other works [9, 13, 36] also explore the programming abstraction for productivity. We consider these directions complementary to FILCO and leave them as future work.

# 6 Conclusion

In this paper, we propose the FILCO architecture and framework to provide a flexible composing architecture that can effectively improve hardware efficiency on hugely diverse workloads. We will keep on exploring FILCO in more diverse scenarios in future works.

**ACKNOWLEDGEMENTS** – This work is supported in part by Brown University New Faculty Start-up Grant, NSF awards #2140346, #2231523, #2441179, #2348306, #2511445, #2518375, #2536952. We thank AMD for the hardware and software donations.

## References


[1] Mohamed S Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, and Andrew C Ling. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *2018 28th international conference on field programmable logic and applications (FPL)*. IEEE, 411–4117.

[2] AMD/Xilinx. 2021. Versal Adaptive Compute Acceleration Platform. https://www.xilinx.com/products/silicon-devices/acap/versal.html.

[3] AMD/Xilinx. 2023. *AI Engine API and Intrinsics User Guide*.

[4] AMD/Xilinx. 2023. *Versal ACAP AI Engine System C Simulator*.

[5] Mohammed S Bensaleh, Yaman Sharaf-Dabbagh, Hazem Hajj, Mazen AR Saghir, Haitham Akkary, Hassan Artail, Abdulfattah M Obeid, and Syed Manzoor Qasim. 2018. Optimal task scheduling for distributed cluster with active storage devices and accelerated nodes. *IEEE Access* 6 (2018), 48195–48209.

[6] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. 2023. Inter-layer scheduling space definition and exploration for tiled accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[7] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. GMT: GPU Orchestrated Memory Tiering for the Big Data Era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 464–478. doi:10.1145/3620666.3651353

[8] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. 2024. Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference. *ACM Trans. Reconfigurable Technol. Syst.* 18, 1, Article 5 (Dec. 2024). doi:10.1145/3656177

[9] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A programming model for composable accelerator design. *Proceedings of the ACM on Programming Languages* (2024).

[10] Peiyan Dong, Jinming Zhuang, Zhuoping Yang, Shixin Ji, Yanyu Li, Dongkuan Xu, Heng Huang, Jingtong Hu, Alex K. Jones, Yiyu Shi, Yanzhi Wang, and Peipei Zhou. 2024. EQ-ViT: Algorithm-Hardware Co-Design for End-to-End Acceleration of Real-Time Vision Transformer Inference on Versal ACAP Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 3949–3960. doi:10.1109/TCAD.2024.3443692

[11] Mathew Hall and Vaughn Betz. 2020. HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs. *arXiv preprint arXiv:2007.10451* (2020).

[12] Zifan He, Anderson Truong, Yingqi Cao, and Jason Cong. 2025. InTAR: Inter-Task Auto-Reconfigurable Accelerator Design for High Data Volume Variation in DNNs. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 123–132.

[13] Erika Hunhoff, Joseph Melber, Kristof Denolf, Andra Bisca, Samuel Bayliss, Stephen Neuendorffer, Jeff Fifield, Jack Lo, Pranathi Vasireddy, Phil James-Roxby, and Eric Keller. 2025. Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 85–94. doi:10.1109/FCCM62733.2025.00043

[14] Shixin Ji, Xingzhen Chen, Jinming Zhuang, Wei Zhang, Zhuoping Yang, Sarah Schultz, Yukai Song, Jingtong Hu, Alex Jones, Zheng Dong, and Peipei Zhou. 2025. ART: Customizing Accelerators for DNN-Enabled Real-Time Safety-Critical Systems. In *Proceedings of the 2025 ACM Great Lakes Symposium on VLSI*.

[15] Shixin Ji, Zhuoping Yang, Xingzhen Chen, Wei Zhang, Jinming Zhuang, Alex K Jones, Zheng Dong, and Peipei Zhou. 2025. DERCA: DetERministic Cycle-Level Accelerator on Reconfigurable Platforms in DNN-Enabled Real-Time Safety-Critical Systems. In *The 46th IEEE Real-Time Systems Symposium, 2025*.

[16] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 71–83. doi:10.1109/HPCA51647.2021.00016

[17] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 71–83.

[18] Jun Liu, Shulin Zeng, Li Ding, Widyadewi Soedarmadji, Hao Zhou, Zehao Wang, Jinhao Li, Jintao Li, Yadong Dai, Kairui Wen, Shan He, Yaqi Sun, Yu Wang, and Guohao Dai. 2025. FlightVGM: Efficient Video Generation Model Inference with Online Sparsification and Hybrid Precision on FPGAs *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA.

[19] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. 2017. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 325–339. doi:10.1145/3575693.3575748

[21] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. 2021. MLP-Mixer: An all-MLP Architecture for Vision. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 24261–24272.

[22] Jianming Tong, Anirudh Itagi, Prasanth Chatarasi, and Tushar Krishna. 2024. FEATHER: A Reconfigurable Accelerator with Data Reordering Support for Low-Cost On-Chip Dataflow Switching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1109/ISCA59077.2024.00024

[23] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*. PMLR.

[24] Chengyue Wang, Xiaofan Zhang, Jason Cong, and James C Hoe. 2025. Reconfigurable Stream Network Architecture. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1848–1866.

[25] Qinggang Wang, Long Zheng, Zhaozeng An, Shuyi Xiong, Runze Wang, Yu Huang, Pengcheng Yao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2024. A scalable, efficient, and robust dynamic memory management library for HLS-based FPGAs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 437–450.

[26] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. arXiv:1907.10701 [cs.LG]

[27] Linus Y. Wong, Jialiang Zhang, and Jing (Jane) Li. 2023. DONGLE: Direct FPGA-Orchestrated NVMe Storage for HLS. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 3–13. doi:10.1145/3543622.3573185

[28] Hanchen Yang, Zishen Wan, Ritik Raj, Joongun Park, Ziwei Li, Ananda Samajdar, Arijit Raychowdhury, and Tushar Krishna. 2025. NSFlow: An End-to-End FPGA Framework with Scalable Dataflow Architecture for Neuro-Symbolic AI. *arXiv preprint arXiv:2504.19323* (2025).

[29] Zhuoping Yang, Jinming Zhuang, Xingzhen Chen, Alex Jones, and Peipei Zhou. 2025. AGILE: Lightweight and Efficient Asynchronous GPU-SSD Integration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1028–1042.

[30] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei Zhou. 2023. AIM: Accelerating Arbitrary-precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP. In *ICCAD*.

[31] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. 2024. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 223–234. doi:10.1145/3626202.3637562

[32] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 27–42.

[33] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[34] Peipei Zhou. 2019. *Modeling and Optimization for Customized Computing: Performance, Energy and Cost Perspective*. Ph. D. Dissertation. University of California, Los Angeles. https://escholarship.org/uc/item/6g7663zw ProQuest ID: Zhou_ucla_0031D_18150; Merritt ID: ark:/13030/m5dk0j3x.

[35] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *FPGA* (Monterey, CA, USA). ACM, 153–164. doi:10.1145/3543622.3573210

[36] Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. 2025. ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA. doi:10.1145/3706628.3708870

[37] Jinming Zhuang, Zhuoping Yang, Shixin Ji, Heng Huang, Alex K. Jones, Jingtong Hu, Yiyu Shi, and Peipei Zhou. 2024. SSR: Spatial Sequential Hybrid Architecture for Latency Throughput Tradeoff in Transformer Acceleration. In *FPGA*. ACM. https://doi.org/10.1145/3626202.3637569